Utiliser SQLite à travers .NET

Par Matthieu Anceret

Date de publication : 5 septembre 2017

Dans le cadre de plusieurs projets, j'ai été amené à utiliser une base de données SQLite dans un contexte .NET. C'est une technologie très intéressante dans le cas où l'on a besoin d'embarquer un stockage de données simple et léger dans une application cliente. Je souhaite revenir avec vous sur le fonctionnement de cette technologie et son utilisation avec l'ORM SQLite.NET.

Commentez

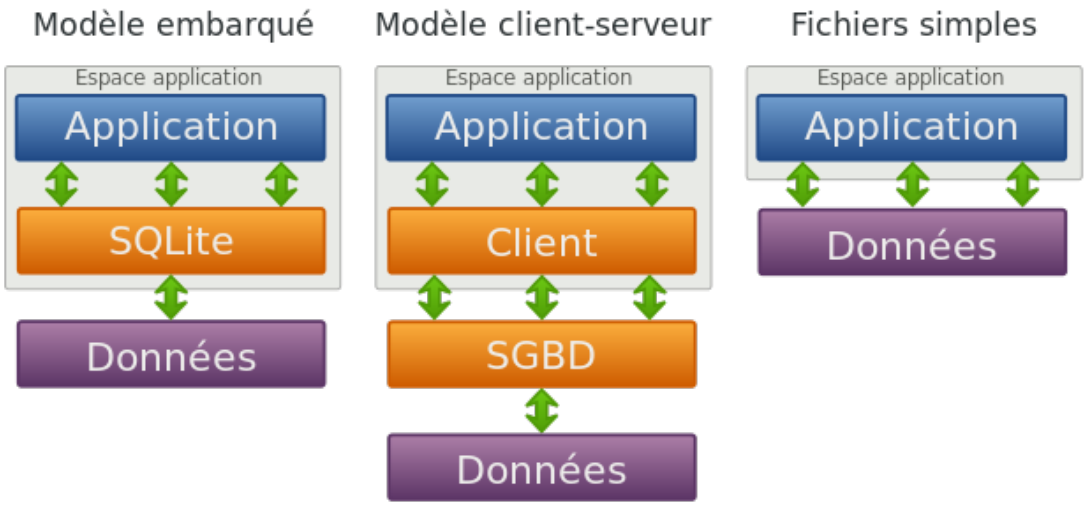
Copyright ® 2017 Matthieu Anceret. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://anceret-matthieu.developpez.com/tutoriels/utiliser-sqlite-a-traversnet/

I - Qu'est-ce que SQLite ?

SQLite est un moteur de base de données relationnelles écrit en C dans les années 2000 par Richard Hipp. Sa principale différence par rapport aux SGBD standards est son fonctionnement en mode local et non en mode client/serveur. Grâce à sa présence sur de nombreuses plateformes, c'est le moteur de base de données le plus utilisé au monde. On peut citer, par exemple les OS pour smartphone (iOS, Android, Symbian…), des logiciels grands publics (Firefox, Chrome, Skype, Evernote, Adobe Photoshop Lightroom, DropBox…) ainsi que dans des bibliothèques standards de langages comme PHP ou Python (retrouvez la liste des principales entreprises sur **ce lien**).

I-A - Une base de données embarquée

Comme indiqué ci-dessus, SQLite ne fonctionne plus sur le paradigme client/serveur, et est donc parfaitement autonome. Le moteur de base de données est inclus dans la bibliothèque, qui elle-même est directement intégrée dans l'application (bibliothèque compatible avec énormément de cibles étant donné qu'il suffit d'avoir à disposition un compilateur C-ANSI). L'ensemble des données est stocké dans un unique fichier, qui a la particularité d'être indépendant du système. Ce fichier contient l'ensemble des éléments de la base (tables, index, données…) et l'accès aux données se fait simplement via l'ouverture du fichier correspondant. SQLite offre une bonne alternative à l'utilisation de fichiers textes, en apportant un cadre et une structure à la gestion de nos données tout en conservant un haut niveau de performances. Il existe également une rétrocompatibilité entre chaque version majeure, ce qui est rassurant pour la pérennité de l'applicatif.



Comparaison des modèles de gestion de données (https://fr.wikipedia.org/wiki/SQLite)

À propos des performances justement, étant donné que l'on n'a pas besoin de serveur, on supprime mécaniquement la latence réseau induite par les architectures de type client/serveur. On est à peu près au même niveau que des simples lectures/écritures sur le système de fichiers. Il est même possible de ne pas utiliser de fichier et de stocker l'ensemble des données en mémoire vive.

Cette architecture a de très nombreux avantages, mais pose tout de même quelques soucis :

- dès qu'un utilisateur débute une opération d'écriture, la lecture est verrouillée (et inversement) et l'utilisateur suivant sera mis en attente. Les accès simultanés par de nombreux utilisateurs peuvent donc poser des problèmes de performances ;
- contrairement au mode client/serveur, il est beaucoup plus compliqué de mettre en place des mécanismes de répartition de charge ;
- il n'est pas possible de découper le fichier d'une base SQLite, ce qui peut être problématique lors de l'utilisation de grosses bases de données sur certains systèmes de fichiers (je pense notamment à la limitation de 4 Go par fichier sur les systèmes FAT32, mais ça laisse quand même pas mal de marge).

Copyright ® 2017 Matthieu Anceret. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://anceret-matthieu.developpez.com/tutoriels/utiliser-sqlite-a-traversnet/

Attention tout de même, les inconvénients que je viens de citer sont à pondérer. Il s'agit en effet de problématiques qui sortent des cas d'utilisation standards pour lesquels SQLite a été imaginé. SQLite est plutôt destiné à être utilisé dans des scenarii où la taille des données est raisonnable (application cliente desktop ou application mobile) et où la centralisation des données est effectuée par un autre biais (service de synchronisation vers un SGBD standard, par exemple). Il est particulièrement adapté dans le cas des applications ayant besoin d'un stockage local pour mode hors-ligne.

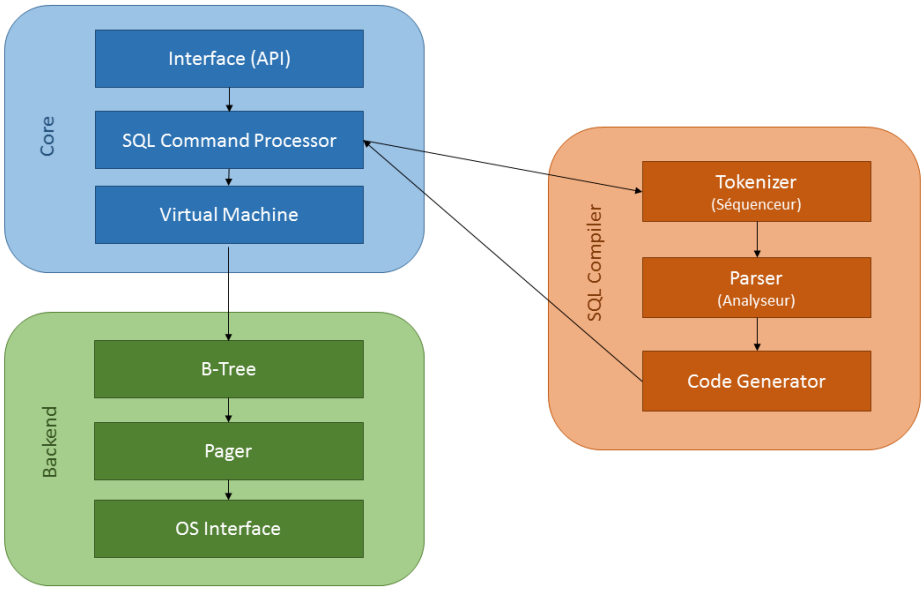
Dernier point avant de passer à la suite, SQLite ne propose pas de gestion des droits (GRANT/REVOKE). La seule possibilité est de gérer les droits directement au niveau du fichier via les mécanismes natifs du système d'exploitation.

I-A-1 - Fonctionnement

Nous allons maintenant voir comment, à partir d'une requête SQL, le moteur se charge de réaliser l'exécution et de retourner les résultats demandés :

1. **L'interface** est le point d'entrée dans la bibliothèque SQLite. Elle permet d'accéder à l'ensemble des méthodes publiques ;
2. C'est ensuite le **processeur de commande SQL** qui prend la main pour réaliser les étapes de compilation de la requête SQL :
    1. Le **tokenizer** (séquenceur) a pour rôle de diviser la requête en *tokens* (jetons) et de les transmettre un par un au *parser (analyseur)*. Il va pour cela se baser sur les mots-clés du langage (SELECT, FROM, GROUP BY…),
    2. À partir des *tokens* reçus, le **parser** va se charger de vérifier leur syntaxe et de leur donner un sens en fonction du contexte,
    3. Enfin, à partir de l'ensemble des *tokens* générés, le **générateur de code** va produire le *bytecode* (code intermédiaire) correspondant ;
3. Le *bytecode* produit est exécuté par la **machine virtuelle** SQLite pour réaliser le travail demandé par la requête SQL ;
4. La machine virtuelle va interpréter les instructions du *bytecode* en faisant appel au **gestionnaire d'arbres**. En effet, c'est de cette façon que SQLite représente les données contenues dans la base de données (un arbre pour chaque table et chaque index) ;
5. Le gestionnaire d'arbres va chercher les données dans les pages du disque via le **pager**. C'est ce dernier qui est chargé des opérations de *rollback/commit* (gestion de l'atomicité) ainsi que du verrouillage du fichier ;
6. Enfin, l'accès effectif aux données est réalisé via des objets VFS (**Virtual File System**). Ces objets permettent le fonctionnement *(*multiplateforme*)*, en fournissant des méthodes pour l'ouverture, la fermeture, la lecture et l'écriture propres à chaque système.

Copyright ® 2017 Matthieu Anceret. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://anceret-matthieu.developpez.com/tutoriels/utiliser-sqlite-a-traversnet/



SQLite - Architecture interne (basé sur le schéma de https://www.sqlite.org/arch.html)

SQLite utilise un typage dynamique, c'est-à-dire qu'au moment de la création de la table, un type dit « d'affinité » est déterminé. Lors de l'insertion, SQLite essaye de convertir la donnée vers ce type, si c'est possible. Quoi qu'il arrive, c'est toujours la cellule qui est chargée de déterminer le type « final » qui correspond à sa représentation en mémoire. À partir de la version 3, SQLite utilise des types différents pour le stockage des données et leur représentation en mémoire (avant cette version, tout était stocké sous forme de chaîne de caractères). Voilà la liste des types disponibles :

- NULL ;
- INTEGER ;
- REAL ;
- TEXT ;
- BLOB.

Attention, SQLite ne propose pas de gestion native des dates. Il est néanmoins possible de les stocker sous la forme de chaîne de caractères (format ISO 8601) ou d'entiers (timestamp), et de les manipuler ensuite via **des fonctions natives**.

SQLite gère bien évidemment les contraintes (PRIMARY KEY, UNIQUE, FOREIGN KEY, NOT NULL, DEFAULT…). Attention tout de même, le support des FOREIGN KEY est géré à partir de la version 3.6.19 et n'est pas activé par défaut. Du côté des déclencheurs (TRIGGERS), le support est lui aussi présent, même chose pour les vues (en lecture seule uniquement) ainsi que pour les tables temporaires et les index (simples ou multicolonnes). Par contre, il n'existe pas de support pour les procédures stockées (mais ce n'est pas forcément gênant au vu de la cible de SQLite).

Du côté des transactions, SQLite va en créer une pour chaque requête visant à modifier la base de données (transaction implicite avec *autocommit* [autovalidation]). Ce fonctionnement permet de garantir l'atomicité des requêtes (mais peut diminuer les performances ; cf. **le chapitre à ce sujet**).

I-A-2 - Configuration

Pour terminer ce chapitre, je vais revenir sur un sujet important : la configuration. De base, SQLite ne nécessite pas de configuration particulière. Mais il est tout de même possible de paramétrer plus ou moins finement certains points de notre base de données via les directives PRAGMA.

Copyright ® 2017 Matthieu Anceret. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://anceret-matthieu.developpez.com/tutoriels/utiliser-sqlite-a-traversnet/

- Gestion des index automatiques :

```
PRAGMA automatic_index = true/false
```

- Gestion de la case lors de l'utilisation du « LIKE » dans une requête :

```
PRAGMA case_sensitive_like = true/false
```

- Activation/désactivation des clés étrangères :

```
PRAGMA foreign_keys = true/false
```

- Passage de la base en lecture seule :

```
PRAGMA query_only = true/false
```

- Gestion de l'encodage :

```
PRAGMA encoding = « UTF-8/16/16le/16be »
```

- Opérations de nettoyage et d'optimisation des données :

```
PRAGMA auto_vacuum / PRAGMA optimize / PRAGMA shrink_memory
```

- Gestion du fichier des transactions (plus de détails dans le **chapitre sur les performances**).
- La liste de l'ensemble des directives PRAGMA est disponible **ici**.

II - Utilisation de « l'ORM » SQLite.NET

L'usage de la bibliothèque SQLite « brute » dans un projet .NET n'est pas forcément l'idéal. Pour faciliter son utilisation, il existe des bibliothèques qui permettent de simplifier son usage. Celle que je vais vous présenter aujourd'hui prend la forme d'un *wrapper* et s'appelle SQLite.NET. Côté installation, rien de plus simple, tout se passe via l'ajout de deux packages NuGet : System.Data.SQLite.Core et SQLite.Net-PCL. Dans les grandes lignes, SQLite.NET permet de gérer, en code managé, les connexions vers une base SQLite et les opérations CRUD, tout en proposant des implémentations synchrones et asynchrones. Il nous permet d'utiliser nos classes C# pour construire nos tables et exécuter des requêtes (pour éviter autant que possible l'écriture de code SQL).

Les exemples de codes ci-dessous ont été effectués à partir d'une application cliente en WPF.

II-A - Récupération de la connexion

La 1re étape à réaliser est de récupérer une connexion vers notre base SQLite (notre fichier donc, si vous avez bien suivi).

```
1. // Récupération du chemin vers notre fichier de base de données
2. string _dbPath = "..\Data\MyDatabase.db3";
3. SQLitePlatformWin32 _platform = new SQLitePlatformWin32();
4.
5. // Instanciation de notre connexion
6. SQLiteConnection connection = new SQLiteConnection(_platform, dbPath);
```

Le 1er paramètre du constructeur de SQLiteConnection correspond à l'implémentation de l'interface ISQLitePlatform propre à la plateforme cible (Win32 dans l'exemple) et sera donc à adapter en fonction du besoin.

Si par hasard, vous deviez définir votre propre implémentation, l'interface ISQLitePlatform contient elle-même quatre interfaces à implémenter :

Copyright ® 2017 Matthieu Anceret. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://anceret-matthieu.developpez.com/tutoriels/utiliser-sqlite-a-traversnet/

- **ISQLiteApi** : fournit l'implémentation aux méthodes natives du moteur SQLite via la directive DLLImport (Open, Close, LibVersionNumber, Initialize, Prepare2, LastInsertRowid…). La plupart de ces méthodes sont dans la DLL SQLite.Interop.dll. C'est ce que l'on appelle l'interopérabilité ;
- **IStopwatchFactory** : fournit une implémentation de la classe Stopwatch sous forme de *factory* ;
- **IReflectionService** : fournit deux méthodes (GetPublicInstanceProperties et GetMemberValue) permettant de faire de la réflexion sur des types et des objets ;
- **VolatileService** : fournit une implémentation pour une méthode Write permettant d'écrire sur des champs volatiles (en C#, le mot-clé volatile permet de s'assurer que la valeur de la variable sur laquelle on travaille est bien la dernière version disponible ; c'est notamment utile dans une utilisation *multithread*).

Heureusement (), il est peu probable que vous ayez besoin d'écrire votre propre implémentation, car par défaut, SQLite.NET propose plusieurs implémentations pour la majorité des plateformes : OSX, Win32, WinRT, Windows Phone 8/8.1, Android et iOS.

II-A-1 - Création des tables à partir de notre modèle

Une fois connecté à notre base de données, il est désormais temps de créer les tables. SQLite.NET permet de réaliser cette opération directement à partir de nos classes C# et des annotations sont disponibles pour personnaliser la création des tables.

```
1. [Table("People")]
2. public class Personne
3. {
4.     [PrimaryKey, AutoIncrement]
5.     public int Id { get; set; }
6.     [Column("Name")]
7.     public string LastName { get; set; }
8.     [MaxLength(50)]
9.     public string FirstName { get; set; }
10.     [Indexed]
11.     public int RoleId { get; set; }
12. }
13.
14. public class Role // Je ne spécifie pas de nom, la table prendra donc le nom de la classe
15. {
16.     [PrimaryKey, AutoIncrement]
17.     public int Id { get; set; }
18.     public string Name { get; set; }
19.     [Ignore] // J'ai choisi de ne pas stocker ce champ en BDD
20.     public bool IsUsed { get; set; }
21. }
```


*Attention, pour les clés primaires, si l'on souhaite profiter de l'auto-incrément, il faut utiliser le type **int** (et il ne sera donc pas possible de faire des clés primaires composites).*

Une fois notre modèle écrit et annoté, il ne nous reste plus qu'à lancer la création des tables.

```
1. SQLiteConnection connection = new SQLiteConnection("myDb.db3");
2. connection.CreateTable<Role>();
3. connection.CreateTable<Personne>();
4.
5. // Utilisation de l'API asynchrone
6. SQLiteAsyncConnection connAsync = new SQLiteAsyncConnection("myDb.db3");
7. connection.CreateTableAsync<Role>();
8. connection.CreateTableAsync<Personne>();
```

L'API SQLite.NET est disponible en version synchrone et asynchrone (en général les méthodes sont les mêmes, seul change le mot clé « async » à la fin des noms de méthode). Cette dernière utilise la TPL (Task Parallel Library) et retourne des « Task » permettant l'usage des mots-clés « async »/« await ».

Copyright ® 2017 Matthieu Anceret. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://anceret-matthieu.developpez.com/tutoriels/utiliser-sqlite-a-traversnet/

II-A-2 - Opérations CRUD

Une fois notre modèle en place, on peut commencer à réaliser des opérations CRUD.

```
1. public void Test(SQLiteConnection conn)
2. {
3.     Role r1 = new Role() { Name = "Administrator" };
4.     Role r2 = new Role() { Name = "Visitor" };
5.     r1 = conn.Insert(r1);
6.     r2 = conn.Insert(r2);
7.     Personne p1 = new Personne() { LastName = "Pierre", FirstName = "Paul", RoleId = r1.Id };
8.     conn.Insert(p1);
9.     // Des méthodes similaires existent pour les opérations Update et Delete
10.     List<Role> roles = conn.Table<Role>().Where(x => x.Name == "Administrator").ToList();
11.     IEnumerable<Personne> personnes = conn.Query<Personne>("SELECT * FROM People WHERE RoleId
 = {0}", r1.Id);
12. }
```

L'utilisation de requêtes écrites directement en SQL (via la méthode **Query(…)**) nous fait perdre les avantages d'utiliser C# (pas de vérification syntaxique à la compilation, par exemple ; il faut donc faire attention à bien repasser sur ce code en cas de modification du modèle), mais permet de réaliser des requêtes complexes de manière efficace et performante. Dans le cas d'une requête sur une table avec de nombreuses colonnes, on peut écrire du code qui nous retourne uniquement les champs désirés (équivalent de SELECT x, y, z au lieu de SELECT *). C'est aussi très utile dans le cas où il est nécessaire de faire plusieurs jointures entre différentes tables (évite dans ce cas l'utilisation de boucle ou le passage par des objets C# pour stocker nos résultats intermédiaires).

Il est aussi possible d'utiliser la méthode **Execute(…)** qui permet d'exécuter du code SQL sans retour de données (contrairement à **Query(…)**). C'est particulièrement utile dans le cas de suppressions ou de mises à jour en masse (au lieu d'utiliser une boucle). Pour les insertions en masse, il existe aussi la méthode **InsertAll(IEnumerable, bool runInTransaction)** ainsi que son équivalent **UpdateAll** pour les mises à jour.

II-A-3 - Transaction

Il est aussi possible de gérer l'exécution de code SQL dans une transaction explicite (c'est une bonne pratique pour optimiser les performances dans certains cas ; j'aborde le sujet un peu plus bas).

```
1. // Méthode 1
2. SQLiteConnection conn = new SQLiteConnection("myDb.db3");
3. db.RunInTransaction(() =>
4. {
5.     conn.Insert(role);
6.     conn.Insert(personne);
7. });
8.
9. // Méthode 2
10. try
11. {
12.     conn.BeginTransaction();
13.     string statement = "SELECT * FROM Role";
14.     conn.Execute(statement);
15.     conn.Commit();
16. }
17. catch(Exception ex)
18. {
19.     if (conn != null && conn.IsInTransaction)
20.     {
21.         conn.Rollback();
22.     }
23. }
```


Copyright ® 2017 Matthieu Anceret. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://anceret-matthieu.developpez.com/tutoriels/utiliser-sqlite-a-traversnet/

III - SQLite-Net Extensions

Si vous avez besoin d'aller plus loin dans la gestion des données de votre base, il existe les SQLite-Net Extensions. Ces dernières permettent de gérer de manière automatique les relations entre instances : One-To-One, One-To-Many, Many-To-One et Many-To-Many. La mise en place passe uniquement par l'ajout d'attributs sur vos classes et il n'y a aucune modification du schéma de base de données (pas de création de table d'association, par exemple). Tout se passe à l'exécution par le biais de la réflexion.

Prenons un exemple simple avec une entité représentant un compteur (eau, gaz, électricité…) que nous appellerons « Compteur » (!) et une entité pour stocker les mesures prises à intervalles réguliers pour chaque compteur que nous appellerons « Mesure » :

```
1. public class Compteur
2. {
3.     [PrimaryKey, AutoIncrement]
4.     public int Id { get; set; }
5.     public string Nom { get; set; }
6.         public TypeCompteur Type { get; set; }
7. }
8.
9. public class Mesure
10. {
11.     [PrimaryKey, AutoIncrement]
12.     public int Id { get; set; }
13.     [Indexed]
14.     public int CompteurId { get; set; }
15.     public DateTime Date { get; set; }
16.     public double Value { get; set; }
17. }
```

Pour retrouver facilement les mesures de notre compteur, il faut exécuter la requête suivante :

```
return db.Query<Mesure> ("SELECT * FROM Mesure WHERE CompteurId=?", compteur.Id);
```

Il est dommage de ne pas pouvoir les retrouver en passant directement par notre objet compteur. Nous allons donc modifier notre modèle pour prendre en charge les relations :

```
1. public class Compteur
2. {
3.     [PrimaryKey, AutoIncrement]
4.     public int Id { get; set; }
5.     public string Nom { get; set; }
6.     public TypeCompteur Type { get; set; }
7.     [OneToMany(CascadeOperations = CascadeOperation.All), ReadOnly = true]
8.     public List<Mesure> Mesures { get; set; }
9. }
10.
11. public class Mesure
12. {
13.     [PrimaryKey, AutoIncrement]
14.     public int Id { get; set; }
15.     [ForeignKey(typeof(Compteur))]
16.     public int CompteurId { get; set; }
17.     [ManyToOne]
18.     public Compteur Compteur { get; set; }
19.     public DateTime Date { get; set; }
20.     public double Value { get; set; }
21. }
```

Grâce à ces attributs, SQLite fait désormais le lien entre nos deux entités, et il est plus facile de parcourir ou de créer l'arbre complet d'une entité et de ses enfants :

```
1. // Création du compteur
2. Compteur compteur = new Compteur() {
```


Copyright ® 2017 Matthieu Anceret. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://anceret-matthieu.developpez.com/tutoriels/utiliser-sqlite-a-traversnet/

```
    Nom = "Linky",
    Type = TypeCompteur.Electricite
};
db.Insert(compteur);

// Création d'une mesure
Mesure mesure = new Mesure() {
    Date = DateTime.Now,
    Value = 15.2,
};
db.Insert(mesure);

// Association de la mesure avec le compteur
compteur.Mesures = new List<Mesure> { mesure };
// Màj en base
db.UpdateWithChildren(compteur);

// Vérification de l'association retour
if (mesure.Compteur == compteur) {
    // OK !
}

// Récupération d'une mesure avec chargement automatique des propriétés liées
Mesure storedMesure = db.GetWithChildren<Mesure>(mesure.Id);
if (compteur.Nom.Equals(storedMesure.Compteur.Nom)) {
    // OK !
}
```

Pour simplifier l'écriture, il est possible d'utiliser la syntaxe suivante :

```
Mesure mesure = new Mesure() {
    Date = DateTime.Now,
    Value = 15.2,
};

Compteur compteur = new Compteur() {
    Nom = "Linky",
    Latitude = 0.0,
    Longitude = 0.0,
    Mesures = new List<Mesure> { mesure }
};

db.InsertWithChildren(compteur);
```

Dans les exemples ci-dessus, j'utilise des clés primaires en auto-incrément. Si ce n'était pas le cas, il faudrait bien penser à valoriser les clés primaires lors de la création des entités pour que SQLite puisse faire les liaisons.

Petite spécificité pour les relations de type **« Many-To-Many »**, il est nécessaire de créer une entité intermédiaire qui va se charger de stocker les couples d'*id*. Cette entité ne sera jamais directement utilisée dans le code, ni créée en base de données. En reprenant nos deux entités ci-dessus, et en imaginant qu'une mesure puisse être associée à plusieurs compteurs, voici la classe supplémentaire qu'il faudrait écrire :

```
public class CompteurMesure
{
    [ForeignKey(typeof(Compteur))]
    public int CompteurId{ get; set; }
    [ForeignKey(typeof(Mesure))]
    public int MesureId{ get; set; }
}
```

Par défaut, les opérations CRUD ne sont pas récursives, mais la plupart peuvent être configurées comme telles. Il suffit de spécifier, dans les paramètres des attributs, la propriété **« CascadeOperations »** avec une ou plusieurs des valeurs suivantes : CascadeRead, CascadeInsert (à utiliser en parallèle des méthodes de type InsertWithChildren) et CascadeDelete. Ces opérations peuvent être combinées avec le caractère « **|** » ou utilisées toutes à la fois avec **CascadeOperation.All**.

Copyright ® 2017 Matthieu Anceret. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://anceret-matthieu.developpez.com/tutoriels/utiliser-sqlite-a-traversnet/

Pour finir sur le sujet des relations, il me reste encore à présenter un attribut très pratique : **ReadOnly**. Celui-ci permet de spécifier qu'une propriété n'est pas concernée par les opérations d'insertions, de modifications ou de suppressions. Il est particulièrement utile dans les cas des propriétés inversées des relations ManyToOne et ManyToMany. Je l'ai utilisé sur mon exemple avec les compteurs et les mesures sur la propriété « Mesures » de l'entité « Compteur », pour empêcher l'ajout direct d'une nouvelle mesure à partir du compteur.

Enfin, SQLite-Net Extensions apporte le support des colonnes de type **« text-blobbed »**. Ce type de colonne permet de stocker facilement un objet simple (une liste ou un dictionnaire, par exemple) dans une colonne d'une table. SQLite va tout simplement mettre en place un mécanisme de sérialisation/désérialisation automatique dans une colonne de type texte. Par défaut, c'est un sérialiseur JSON qui est utilisé (Newtonsoft JSON.Net).

IV - Gestion de l'upgrade d'une base de données

Une fois notre v1.0 développée et publiée, il est tant de penser à la v2.0. À ce moment-là se pose la question de l'évolution de notre modèle, et donc de notre base de données.

La 1re bonne pratique à mettre en place est de stocker le numéro de version de notre base de données. Il existe pour cela une API spécifique à la plateforme SQLite :

```
1. // Récupération du numéro de version courant
2. string version = sqLiteAsyncConnection.ExecuteScalar<string>("PRAGMA user_version");
3.
4. // Mise à jour du numéro de version
5. sqLiteAsyncConnection.ExecuteScalar<string>("PRAGMA user_version=2;");
```

SQLite propose de réaliser automatiquement la migration d'une table en réutilisant la méthode `CreateTable<T>()`. Attention, cela ne fonctionnera que dans le cas de l'ajout de nouvelles colonnes. En effet, les colonnes qui n'existent plus ne sont pas supprimées (il faut le faire à la main ou laisser tel quel). Cette méthode ne gère pas non plus le changement du type de colonne.

On peut aussi gérer à la main le mécanisme :

```
1. public override void UpgradeDbversion(SQLiteConnection connection, int from, int to)
2. {
3.     if (from == 0)
4.     {
5.         connection.CreateTable<Table1>();
6.         connection.CreateTable<Table2>();
7.         // Maj de la version de BDD
8.         _dbVersion = 1;
9.         from = _dbVersion;
10.         connection.ExecuteScalar<int>(string.Format("PRAGMA user_version = {0}", dbVersion));
11.     }
12.     if (from == 1)
13.     {
14.         connection.CreateTable<MyNewTable>();
15.         connection.CreateTable<Table1>(); // Màj via le mécanisme natif
16.         _dbVersion = 2;
17.         from = _dbVersion;
18.         connection.ExecuteScalar<int>(string.Format("PRAGMA user_version = {0}", dbVersion));
19.     }
20.     if (from == 2)
21.     {
22.         // Màj "à la main"
23.         connection.ExecuteScalar<int>("ALTER TABLE Table1 ADD COLUMN NewColumn
 varchar(36);");
24.         connection.Execute("ALTER TABLE Table1 RENAME TO OLDTable1;");
25.         ...
26.     }
27.     ...
28. }
```


Copyright ® 2017 Matthieu Anceret. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://anceret-matthieu.developpez.com/tutoriels/utiliser-sqlite-a-traversnet/

Il est très important de penser à conserver un fichier pour chaque version de la base de données (au minimum celui de la v1). Cela permet de réaliser des tests de migration de bout en bout.

V - Performances

L'un des soucis les plus courants avec SQLite est l'insertion, la mise à jour ou la suppression des données en masse. Généralement, la cause est liée au fait que SQLite crée une transaction implicite pour chaque opération Insert/Update/Delete. La solution à ce problème est simple : il suffit de débuter explicitement une transaction au démarrage du traitement et de la valider (*committer*) une fois l'ensemble des opérations terminées. En effet, si SQLite détecte une transaction en cours, il n'en recrée pas une. Il est aussi possible d'utiliser la méthode **InsertAll(IEnumerable<T>)** qui va avoir le même comportement.

Dans le cas où vous souhaitez insérer un gros volume de données dans une table possédant un ou plusieurs index, il est conseillé de les créer après l'insertion des données. Attention aussi à ne pas en abuser, ils peuvent avoir l'effet inverse de celui escompté et faire baisser les performances. Pour les requêtes avec une clause du type `WHERE MyColumn > 10 ORDER BY Length(MyColumn)`, il est possible de créer un index de cette façon : `CREATE INDEX idx_test ON MyTable(LENGTH(MyColumn))`. Ce type d'index permettra d'éviter de parcourir l'ensemble de la table lors de l'exécution de la requête, mais ne peut être utilisé que si c'est la même colonne qui est dans la clause WHERE et dans la clause `ORDER BY`.

Si vous utilisez un modèle de données complexes et que vous avez besoin de réaliser des traitements impliquant plusieurs tables, je vous conseille d'éviter de passer par l'ORM SQLite-Net et de plutôt écrire vous-même votre code SQL (avec des jointures) que vous exécuterez via un Query ou un Execute. Cela permettra d'améliorer drastiquement les performances.

Par défaut, SQLite est configuré pour maximiser la sécurité, l'intégrité et la robustesse des données et non les performances. Il est possible de modifier cela (à faire en toute connaissance de cause évidemment).

- **PRAGMA TEMP_STORE** : permet de configurer l'emplacement des données temporaires.
    - 0 : valeur par défaut (dépend de la valeur de la variable de préprocesseur SQLITE_TEMP_STORE utilisée lors de la compilation).
    - 1 : dans un fichier.
    - 2 : en mémoire.
- **PRAGMA JOURNAL_MODE** : permet de configurer le fonctionnement du fichier de journal des transactions.
    - DELETE : mode par défaut ; pour chaque transaction, un fichier est créé puis supprimé une fois la transaction validée. SQLite ne peut donc pas gérer plusieurs transactions à la fois, et il ne peut donc pas y avoir plusieurs opérations d'écriture. Les opérations de lecture ne peuvent pas être exécutées en même temps, la base est donc bloquée.
    - TRUNCATE : réutilisation du même fichier pour chaque transaction. On se contente de le vider à chaque fois. Ce mode est généralement plus performant (évite la suppression/création d'un fichier à chaque fois).
    - PERSIST : utilisation d'un même fichier, et effacement seulement du début du fichier à la fin de la transaction.
    - MEMORY : le fichier des transactions est créé en mémoire. Attention à la perte de données en cas de bogue !
    - WAL (Write-Ahead Logging) : ce mode permet l'écriture et la lecture en simultané et promet des gains de performance. Attention, ce mode n'est pas compatible avec les systèmes de fichiers en réseau et est légèrement moins performant en lecture (environ 2 %), au profit d'un gain important en écriture.
    - OFF : désactivation du journal des transactions.
- **PRAGMA SYNCHRONOUS** : permet de configurer le mode de *rollback* du journal des transactions.
    - 0
    - 1

Copyright ® 2017 Matthieu Anceret. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://anceret-matthieu.developpez.com/tutoriels/utiliser-sqlite-a-traversnet/

    - 2
- **PRAGMA LOCKING_MODE**
    - NORMAL
    - EXCLUSIVE
- **PRAGMA CACHE_SIZE** et **PRAGMA PAGE_SIZE**
- **PRAGMA OPTIMIZE** : apparue dans la v3.18.0, permet d'optimiser la base de données. La documentation officielle conseille d'exécuter cette instruction juste avant chaque fermeture de connexion (ou via une programmation dans le cas des applications à longue durée d'utilisation).

Pour finir, il existe une fonction SQLite permettant de « reconstruire » la base de données pour la réorganiser, la réduire et optimiser les performances. Cette commande s'appelle **« VACUUM »**. Pour mieux comprendre le fonctionnement et l'intérêt de ce mécanisme, je préfère vous faire un lien vers **cet article de blog** qui explique très bien ce que c'est et à quoi ça sert.

VI - Conclusion et retours d'expériences

SQLite est une technologie que j'utilise très souvent lors de mes projets professionnels, et dans 95 % des cas, en association avec l'ORM SQLite-Net (en C# donc). C'est donc un outil que je maîtrise plutôt bien et pour lequel j'ai accumulé un certain nombre d'astuces et de bonnes pratiques (activation du mode WAL, écriture des requêtes complexes « en dur », utilisation des transactions explicites…). La majorité de mes utilisations consiste à employer SQLite comme stockage local côté application cliente (client lourd WPF, application Windows 8.1/UWP ou application mobile Android) pour offrir un mode hors-ligne. En parallèle, la partie BackOffice est généralement plus classique, avec le triptyque ASP.Net/EntityFramework/SQL Server couplé à un moteur de synchronisation « maison ».

C'est un produit très mature, mais malgré tout en évolution constante (nouvelles fonctionnalités ainsi que des optimisations de performance). Il est d'ailleurs important de bien penser à mettre à jour les packages NuGet pour profiter des dernières améliorations (pas de risque particulier contrairement à un SGBD classique, mais il est tout de même préférable de bien lire les *releases* notes et faire attention aux *breaking changes*).

VII - Note de la rédaction de Developpez.Com

Nous tenons à remercier **Winjerome** pour la mise au gabarit et **Jacques_Jean** pour la relecture orthographique.

Copyright ® 2017 Matthieu Anceret. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://anceret-matthieu.developpez.com/tutoriels/utiliser-sqlite-a-traversnet/